Parallel Actor Monitors: Disentangling Task-Level Parallelism from Data Partitioning in the Actor Model.

Christophe Scholliers [a,1,*], Éric Tanter[b,2], Wolfgang De Meuter[a]

[a] *Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, Elsene, Belgium*
[b] *PLEIAD Laboratory, DCC University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile*

Abstract

While the actor model of concurrency is well appreciated for its ease of use, its scalability is often criticized. Indeed, the fact that execution *within* an actor is sequential prevents certain actor systems to take advantage of multicore architectures. In order to combine scalability and ease of use, we propose Parallel Actor Monitors (PAM), as a means to relax the sequentiality of intra-actor activity in a structured and controlled way. A PAM is a modular, reusable scheduler that permits to introduce intra-actor parallelism in a local and abstract manner. PAM allows the stepwise refinement of local parallelism within a system on a per-actor basis, without having to deal with low-level synchronization details and locks. We present the general model of PAM and its instantiation in the AmbientTalk language. Benchmarks confirm the expected performance gain.

Keywords: Actors, Concurrency, Monitors, Efficiency

1. Introduction

The actor model of concurrency [1] is well recognized for the benefits it brings for building concurrent systems. Actors are *strongly encapsulated* entities that

*Corresponding author
Email addresses: cfscholl@vub.ac.be (Christophe Scholliers), etanter@dcc.uchile.cl (Éric Tanter), wdmeuter@vub.ac.be (Wolfgang De Meuter)

[1]Funded by a doctoral scholarship of the Institute for the Promotion through Science and Technology in Flanders (IWT-Vlaanderen).
[2]Partially funded by FONDECYT Project 1110051..

communicate with each other by means of asynchronous message passing. Data races are prevented by design in an actor system because data cannot be shared between actors, and actors process messages sequentially. However, these strong guarantees come at a cost: efficiency.

The overall parallelization obtained in an actor system stems from the parallel execution of multiple actors. Upon the reception of a message, an actor executes a task which can only manipulate data local to the actor. Therefore, either data is encapsulated in an actor and only one task can manipulate that data or the data is distributed over multiple actors in order to exploit parallelism. Because of the strong *data-task entanglement* in the actor model, the actor programmer is *forced* to partition her data in order to exploit parallelism. In general, the structure and choice of algorithms strongly depend on the structure of the underlying data [2]. When each actor possesses a partitioning of the data, global operations over the data must be implemented by a well defined protocol between those actors. These protocols have to be carefully encoded to guarantee properties like sequentiality and consistency, important properties that can be taken for granted within a single actor. The actor programmer needs to encode these properties manually as a side effect of trying to exploit parallelism. The implementation of global data operations is low-level, error-prone, and potentially inefficient. Moreover, restructuring large amounts of data at runtime by transferring the data from one actor to another, involves high overheads. In conclusion, data-task entanglement ensures important properties at a local level but at the same time forces the programmer to partition her data when trying to implement data-level coarse grained parallelism [3]. The partitioning of data in order to exploit parallelism leads to complex and inefficient code for global operations when coordination is required between the actors that encapsulate the partitioned data.

Let us further illustrate the problem by means of an example. Consider a group of actors, of which one is a dictionary actor. The other actors are clients of the dictionary: one actor does updates (writer), while the others only consult the dictionary (readers). The implementation of the dictionary with actors is easy because the programmer does not need to be concerned with data races: reads *and* writes to the dictionary are ensured to be executed in mutual exclusion. The programmer is sure that no other actor could have a reference to the data encapsulated in the dictionary actor. The asynchronous nature of the actor model also brings distribution transparency as the dictionary code is independent from the physical location of the actors.

However, when the number of readers increases the resulting application performs badly precisely because of the benefits of serial execution of requests to

the dictionary actor: there are no means to process read-only messages in parallel and thus the dictionary actor becomes the bottleneck. In order to scale the dictionary, the programmer is thus forced to partition his data over a group of dictionary actors, for example one per letter. As explained, this introduces accidental complexity for global operations over the dictionary. A pervious simple operation, such as counting all the words in the dictionary ending at "ing", has to be implemented by a protocol, *e.g.* block all actors that hold parts of the dictionary, ask them to perform this operation on their local data and wait for their answer, and finally release the actors.

The problems of data-task entanglement within the actor model has been acknowledged by many of the current actor implementations as many implementations compromise the strong encapsulation properties for efficiency [4]. However, such solutions are both unsafe and ad-hoc as we show in the next section. In order to disentangle task-level parallelism from data partitioning in a structured and high-level manner we propose the use of *parallel actor monitors* (PAM). In essence, a PAM is a *scheduler* that expresses a coordination strategy for the *parallel execution of messages within a single actor*. Since with a PAM, messages can be processed in parallel within the same actor, the programmer is no longer forced to partition her data in order to exploit parallelism.

There are four main contributions in applying the PAM model in order to solve the problems of data-task entanglement introduced by traditional actor systems.

1. **Efficiency.** A PAM makes it possible to take advantage of parallel computation for improved scalability. Benchmarks of our prototype implementations suggest speedups that are almost linear to the number of processors available (Section 8).
2. **Modularity.** A PAM is a modular, reusable scheduler that can be parameterized and plugged into an actor to introduce intra-actor parallelism without modification of the original code. This allows generic well-defined scheduling strategies to be implemented in libraries and reused as needed. By using a PAM programmers can separate the coordination concern from the rest of the code.
3. **Locality.** Binding a PAM to an actor *only* affects the parallel execution of messages inside that single actor. The scheduling strategy applied by one actor is completely transparent to other actors. This is because a PAM preserves the strong encapsulation boundaries between actors.
4. **Abstraction.** A PAM is expressed at the same level of abstraction as actors: the scheduling strategy realized by a PAM is defined in terms of a message

queue, messages, and granting permissions to execute. A PAM programmer does *not* refer explicitly to threads and locks. It is the underlying PAM system that takes responsibility to hide the complexity of allocating and handling threads and locks for the programmer.

The next section gives an overview of the closest related work and shows why current approaches are not sufficient. Section 3 presents our parallel actor monitors in a general way, independent of a particular realization. Section 5 then overviews how our implementation of PAM on top of AmbientTalk is used to express canonical examples as well as a more complex coordination strategy. Section 8 evaluates PAM against current actor systems, and provides an assessment of the implementation through a set of benchmarks. Section 9 concludes.

2. Related Work

When ABCL [5] introduced state in the previously functional actor model one of the major design decisions for synchronization was the following:

> "*One at a time: An object always performs a single sequence of actions in response to a single acceptable message. It does not execute more than one sequence of actions at the same time.*"

Since then it has been one of the main rules in stateful actor languages such as Erlang [6], Akka and Scala [7], Kilim [8], ProActive [9], E [10], Salsa [11] and AmbientTalk [12]. In all these languages execution of parallel messages within a single actor is disallowed by construction, *e.g.* every actor has only one thread of control and data cannot be shared between actors. As seen before, this leads to scalability issues when resources have to be shared among a number of actors. With this wild growth of actor languages it is not surprising that we are not the first ones to observe that the actor model is too strict [13]. There a number of alternatives used in actor-based systems to overcome this limitation.

First, actor languages that are built on top of a thread-based concurrency system can allow an "escape" to the implementing substrate. For instance, AmbientTalk [12] supports symbiosis with Java [14], which can be used to take advantage of multi-threading as provided by the Java language. However, such a backdoor reintroduces the traditional concurrency problems and forces the programmer to think in two different paradigms (actor model and thread based model).

Another approach is to introduce heterogeneity in the system by allowing actors to coexist with non-encapsulated, shared data structures. This is the case

for instance in the ProActive actor-based middleware for Java [9]. In ProActive "naked" data structures can be created, which actors can access freely, concurrently. Avoiding data races is then done by *suggesting* client actors to request access to the shared resources through a coordinator actor [3].

A major issue with this approach is that the model does not *enforce* clients to use a coordinator actor: nothing prevents an actor to access the shared data structure directly, thereby compromising thread safety. Readers and writers themselves have to notify the coordinator when they are finished using the shared resource; of course, failing to do so results in faulty programs. The introduction of shared data violates both locality and modularity, because the use of shared data influences the whole program; it basically reintroduces the traditional difficulties associated with threads and locks. Similar to Proactive, in Scala Actors, Kilim, JavAct and Jetlang a message can carry objects which are passed by reference instead of by copy, thus introducing shared state between the actors [4].

JCoBox [15] and Creol [16] introduce a generalization of active objects which enables cooperative multi-tasking within an active object. This is opposed to PAM which provides full preemptive threads inside an actor. This means that to enable parallelism in JCoBox or Creol the body of the active object has to be modified, whereas this is not necessary in PAM.

Finally, Akka and Scala [7] provide a from of software transactional memory which allows transactions to be spawned over multiple actors. The use of this system requires intrusive changes inside the body of the actors involved. For example, only transactional references are captured by the system. In contrast A PAM is *modular* abstraction that can be plugged into an actor to introduce intra-actor parallelism without modification of the original code.

Next to actor-based systems other related work deals with the coordination and synchronization of messages in object-oriented systems. Join methods [17] are defined by a set of method fragments and their body is only executed when all method fragments are invoked. Implementation of Join methods can be found in C# [18] and Join Java [19]. Synchronizers [20] is a declarative modular synchronization mechanism to coordinate the access to one or a group of objects by enabling and disabling groups of method invocations. We come back to synchronizers and show their implementation in PAM in Section 5.

In summary, while the actor model has proven to be an appropriate mechanism for concurrent and distributed applications current approaches to deal with

[3] Example code can be found on http://proactive.inria.fr/index.php?page=reader_writers

intra-actor parallelism are both ad-hoc and unsafe. As a solution we present PAM, an abstraction which allows programmers to modularize their intra-actor coordination code in a local and abstract manner.

3. Parallel Actor Monitor

A parallel actor monitor, PAM, is a low-cost, thread-less *scheduler* controlling parallel execution of messages within an actor. In many actor systems, an actor encapsulates a number of passive objects, accessed from other actors through asynchronous method calls. A PAM is therefore a *passive object* that controls the synchronization aspect of objects living within an actor, whose functional code is not tangled with the synchronization concern.

It is our objective to bring the benefits of the model of *parallel object monitors* (POM) [21] to actor programming. It is therefore unsurprising that the operational description and guarantees of a parallel actor monitors closely resemble those of POM. POM is formulated in a *thread-based*, synchronous world: PAM is its adaptation to an actor-based, purely asynchronous *message-passing* model.

3.1. Operational Description

A PAM is a monitor with a *schedule method* responsible for specifying how messages in an actor queue should be scheduled, possibly in parallel. A PAM also defines a *leave method* that is executed by each thread once it has executed a message. These methods are essential to the proposed abstraction, making it possible to reuse functional code as it is, adding necessary synchronization constraints externally. An actor system that supports PAM allows the definition of schedulers and the binding of schedulers to actors.

Figure 1 illustrates the operation of a PAM in more detail. The figure displays an actor runtime system hosting a single actor and its associated PAM, as well as a thread pool, responsible for allocating threads to the processing of messages. Several actors and their PAMs can live within the runtime system but at any moment in time a PAM can only be bound to one actor. When an asynchronous call is performed on an object hosted by an actor (1), it is put in the actor queue as a message object (2). Messages remain in the queue until the scheduling method (3) grants them permission to execute (4).

The scheduling method can trigger the execution of several messages. All selected messages are then free to execute *in parallel* with free access to the actor (5), each task runs in a thread allocated by the thread pool of the runtime system. Note that, if allowed by the scheduler, new messages can be dispatched

before a first batch of selected messages has completed. In contrast to the traditional actor model, where messages are executed sequentially [5], a PAM enables the parallel execution of multiple messages.

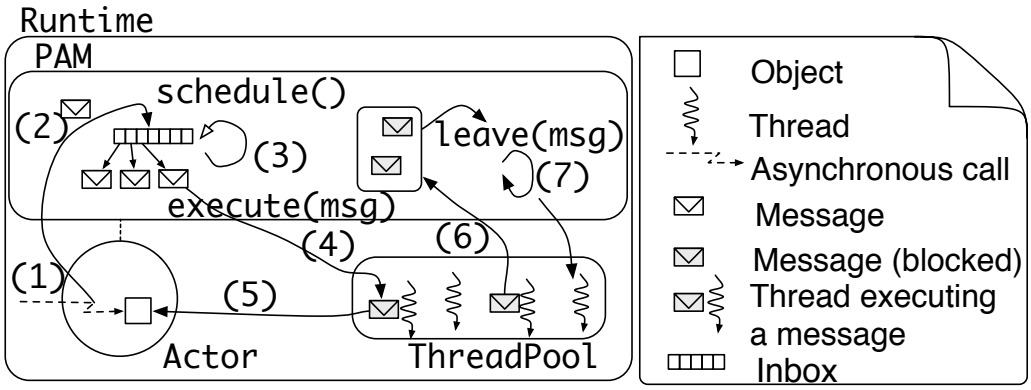

Figure 1: Operational sketch of PAM

Finally, when a thread has finished the execution of its associated message (6), the thread calls the leave method of the PAM (7). To run the leave method, a thread may have to wait for the scheduler monitor to be free (a PAM *is* a monitor): invocations of the scheduling and leaving methods are always safely executed, in *mutual exclusion*. A thread about to leave the monitor will first execute the scheduling method again in case the inbox of the actor is not empty. The fact that a thread spends some time scheduling requests for other threads (recall that the scheduler itself is a passive object) allows for a more efficient scheduling by avoiding unnecessary thread context switches.

One of the main operational differences between PAM and POM is a direct consequence of the underlying paradigm. In POM there are several threads in the system which are being coordinated by the POM while a PAM coordinates messages. Moreover in PAM, the thread of the caller actor cannot be reused, as done in POM, because this would block the caller and violate the asynchronous nature of the model.

3.2. PAM at runtime

To further illustrate the working of a PAM, let us consider an actor whose PAM implements a simple join pattern coordination: when both a message `a` and a message `b` have been received by the actor, both can proceed in parallel. Otherwise, the messages are left in the queue. Figure 2 shows a *thread diagram* of the

scenario. A thread diagram pictures the execution of threads according to time by picturing the call stack, and showing when a thread is active (solid line) or blocked waiting (dotted line). The diagram shows two threads T1 and T2 (from the thread-pool), initially idle, available for the activity of the considered actor. The state of the actor queue is initially empty.

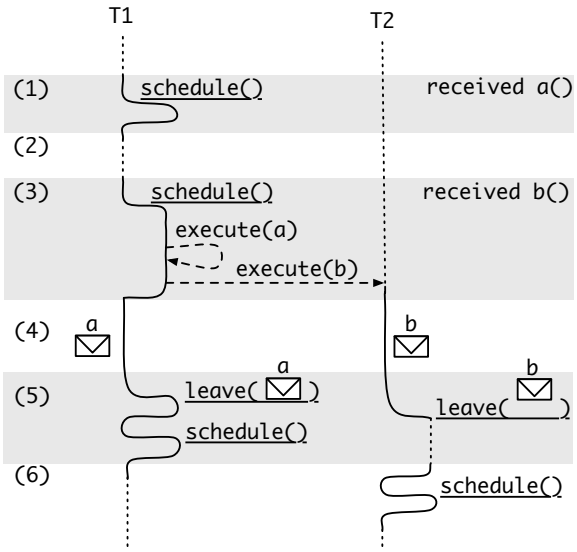

Figure 2: A simple join pattern coordinated by a PAM. *(underlined method calls are performed in mutual exclusion within the scheduler)*

When a message `a` is received in the queue, T1 runs the schedule method (1). Since there is no message `b` in the queue, nothing happens, T1 remains idle (2). When a message `b` is received, T1 runs again the `schedule` method (3). This time, both messages `a` and `b` are found in the queue, so they are both dispatched in parallel. First `a` is dispatched, then `b`. T1 finishes the execution of the schedule method, while T2 starts processing the `b` message. Then, both T1 and T2 are executing, in parallel, their respective messages (4). When T1 finishes processing `a`, T1 calls the `leave` and then the `schedule` method, in mutual exclusion (5). Meanwhile, T2 also finishes processing its message but has to wait until the PAM is free in order to execute both methods itself (6). Note that the `schedule` method is only called at this point if there are pending messages in the queue.

3.3. PAM Guarantees

An actor system supporting PAM should support the following guarantees to the programmers.

1. *An asynchronous message sent to an object encapsulated by an actor that is bound to a PAM is guaranteed to be scheduled by this PAM.*
2. *The schedule and leave methods of a PAM are guaranteed to be executed in mutual exclusion within the PAM, but in parallel with the messages being executed.*
3. *The schedule method is guaranteed to be executed if a message may be scheduled and guaranteed not to be executed when there are no pending messages.*
4. *When the schedule method instructs the execution of a message, the message is executed in parallel with the PAM or as soon as the schedule method is finished.*
5. *When a message has been processed, it is guaranteed that the leave method is called once. Afterwards, the schedule method is only called when the inbox of the PAM is not empty.*

4. AmbientTalk PAM

In this section, we first give a short overview of AmbientTalk, the language that we extended with the PAM coordination mechanism. Second, we show the implementation of PAM in AmbientTalk from the programmer's point of view.

4.1. AmbientTalk in a Nutshell

AmbientTalk [12] is an actor language with an event loop concurrency model adopted from E [10]. In this model, actors are represented as containers of regular objects encapsulating a single thread of execution (*an event loop*) which perpetually take a message from their message queue and invoke the corresponding method of the object denoted as the receiver of the message. The method is then run to completion denoting a *turn*. A turn is executed atomically, *e.g.* an actor cannot be suspended or blocked while processing a message. Only an object's owning actor can directly execute one of its methods. Communication with an object in another actor happens *asynchronously* by means of *far references*: object references that span different actors. In AmbientTalk asynchronous method invocations are indicated by the arrow operator. For example, `O<-m()` denotes an asynchronous message `m` to an object `O`. Upon reception this message is enqueued in the message queue of the actor owning the object O, which eventually processes it in an atomic turn.

scheduler: *codeblock*	executeAll(F)	executeOldestLetter(F)
executeYoungest (F)	executeAllOlderThan(F)	executeAllYoungerThan(F)
executeOlderThan(F_a, F_b)	executeYoungerThan(F_a, F_b)	
executeLetter(L)	listIncomingLetters()	Category()
bindScheduler: S on: A	tagMethod: M with: C_{tag} on: O	contains(F)

Table 1: PAM API

4.2. AmbientTalk PAM API

We extended AmbientTalk in order to allow intra-actor parallelism according to the PAM model presented in the previous sections. The core API of PAM (Table 1) supports abstractions to create and bind a PAM to an actor as well as to coordinate the asynchronous messages sent to an actor. A PAM is created by using the `scheduler:` constructor function, which expects a block of code that implements a scheduling strategy. Every scheduler in AmbientTalk has to implement at least a `schedule` and a `leave` method. The scheduler constructor returns a passive object (the PAM), which can be bound to an actor using the `bindScheduler` construct. After a PAM has been bound to an actor all asynchronous messages sent to this actor are scheduled by the PAM.

Inside the body of the scheduler, the programmer can examine the inbox of the actor, which contains `letters`. Access to the inbox is granted by using `listIncomingLetters()`, which returns the list of letters in the inbox of the actor. A letter contains a `message` and a `receiver`. The receiver is the (passive) object that lives inside the actor to which the message was sent. It is possible to execute a single letter or a group of letters. To start the execution of a single letter, the programmer can apply the `executeLetter` method to a specific letter. Most PAM abstractions however, expect a filter, an object that implements a predicate method `pass` that expects a letter as an argument. The return value of the predicate method `pass` indicates that the letter passed as an argument should be executed or not. For example `executeAll` initiates the execution of all letters in the inbox that pass the filter F. Similarly, to execute only the youngest or oldest letter that passes a filter the programmer can apply the `executeYoungest` or `executeOldest` function respectively.

The `executeOlderThan` function expects two filters and starts the execution of all the messages that pass filter F_a, only if they are older than the first letter that passed F_b. Functions from the API which can only execute one letter return a boolean indicating that a matching letter was executed or not. All the other methods, which can potentially instruct the execution of multiple letters, return an

integer indicating the number of letters they triggered for execution.

4.3. Binding and Reuse

A PAM is a passive object that defines a scheduling strategy. Upon creation it is unbound and does not schedule any message. To be effective, a PAM must be *bound* to an actor with `bindScheduler`. After binding a PAM to a specific actor, all the guarantees listed in section 3.3 hold. For a scheduler to be reusable it has to abstract over the actual message names in order to coordinate the access to the actor. Programmers define PAM in abstract terms by making use of *message categories*, also found in POM. Message categories are constructed by calling the `Category` constructor which creates a category C with its unique category tag C_{tag}. The set of messages belonging to a category is specified by tagging methods with `tagMethod`. A message M belongs to a message category C when it is targeted to invoke a method which is tagged with the category tag C_{tag}. For programmers convenience a method category is also a *filter*. A message category can be used as a filter that only passes letters that contain a message belonging to the message category. By making use of message categories, a reader-writer PAM works not only with a dictionaries whose methods have specific names, but can also be used by actors encapsulating other data structures. The use of categories is further illustrated in the following section.

5. Canonical examples

To illustrate the use of a PAM, we first give the implementation of two classical schedulers and how to bind them to actors in a program. The first example is purely didactic, since it re-introduces sequentiality within an actor: a standard mutual exclusion scheduler. The second example is the classical multiple-reader/single-writer scheduler. Although these examples are easy these two problems clearly show the typical use of PAM. Because PAM is a reincarnation of POM within actors, it allows the implementation of advanced coordination mechanisms, like guards [22] and chords [18, 21]. In this paper we show the PAM implementation of a coordination abstraction called Synchronizers [20].

```
def mutex() { scheduler: {
 def working := false;
 def schedule() {
  if: !(working) then: {
   working := executeOldestLetter();
  };
 };
 def leave(letter) { working := false; };
 };
};
```

Listing 1: Mutual exclusion PAM

```
def dictionaryActor := actor: {
 def dictionary := object: {
  def put() { ... };
  def get() { ... };
 }
}

bindScheduler: mutex() on: dictionaryActor;
```

Listing 2: PAM Binding

5.1. Mutual Exclusion

In this section we show how the normal mutual exclusion behavior of traditional actor systems can be implemented in PAM. Note that this is purely for didactical reasons as mutual exclusion is the default behavior of an actor when no PAM is bound to it. The implementation of such a PAM is shown in Listing 1. The mutual exclusion PAM is created by means of the `mutex` constructor function which returns a new mutex PAM when applied.

The PAM implements the methods `schedule` and `leave` providing mutual exclusion guarantees to the actor. The PAM has one instance variable `working`, initialized to `false` to keep track of whether a message is currently being executed. The `schedule` method only triggers the execution of a message if there are no executing messages already. If so, it executes the oldest letter in the inbox, if any. Its state is updated to the result of invoking `executeOldestLetter`, which indicates if a message was actually triggered or not. The `leave` method changes the state of the scheduler accordingly after a message has been processed. Finally the scheduler is instantiated and bound to the dictionary actor by the `bindScheduler` method as shown in Listing 2.

Note that the definition of the scheduler is simple and does not deal with low-level synchronization and notification details. This is in contrast with conventional monitors where one has to explicitly notify waiting threads. In PAM the underlying system guarantees that after the leave method, the schedule method is automatically invoked if there are waiting letters. Also, note that the actor definition did not suffer any intrusive change.

5.2. Parallel dispatch

The reader-writer scheduler for coordinating the parallel access to a shared data structure is shown in Listing 3. Like before, `RWSched` is a constructor function that, when applied, returns a fresh scheduler. The scheduler defines two method categories, readers (`R`) and writers (`W`). In order to keep track of how

many readers and writers are executing, the scheduler maintains two variables `writing` (boolean) and `readers` (integer). When the scheduler is executing a write letter, no further message can be processed in parallel. In case the scheduler is not executing a write letter, the scheduling method triggers the parallel execution of all the read letters that are older than the oldest write letter, if any, using `executeAllOlderThan`. This call returns the number of dispatched readers, used to update the `readers` state. If there were no reader letters to process (older than the oldest writer), the scheduler dispatches the oldest writer using `executeOldest` . This method returns true if the processing of a letter was actually dispatched, false otherwise. Note that this scheduler uses a fair strategy but could easily be modified to give priority to writers. Finally, the `leave` method updates its state according to which message has finished executing, by either decreasing the number of readers, or turning the `writer` flag to false. To do so it makes use of the `dispatch` construct which re-uses the message category filters in order to decide which code block to execute.

```
def RWSched() { scheduler: {
 def R := Category();
 def W := Category();
 def writing := false;
 def readers := 0;
 def schedule() {
  if: !(writing) then: {
   def executing :=
   super.executeAllOlderThan(R,W);
   readers := readers + executing;
   if: (readers == 0) then: {
       writing := super.executeOldest(W);
   };
  };
 };
  def leave(letter) {
   dispatch: letter as:
   [[R, { readers := readers - 1 }],
    [W, { writing := false }]];
 };
};};
```

Listing 3: Schedule and Leave methods of the Reader/Writer PAM

```
annotateMethod: ‘get with: RWSched.R on: dictionary;
annotateMethod: ‘put with: RWSched.W on: dictionary;
bindScheduler: RWSched() on: dictionaryActor;
```

Listing 4: Instantiating and binding of a PAM to an actor.

In order to use this scheduler with a dictionary actor, one should just annotate the methods of the dictionary with the appropriate categories, and then instantiate and bind the scheduler to the Actor (Listing 4).

5.3. Fair Dining Philosophers

An example where the coordination of multiple objects is essential is the classical Dining Philosophers. In this example there are a number of philosophers which can only do one of two things at a time: eat or think. They are seated around a round table with a fork on their left-hand side and on their right-hand side. In order to eat a philosopher needs to obtain two forks.

In our implementation a philosopher is represented by an object which sends `eat` messages to a table actor. The scheduler which can coordinates the `eat` messages sent to the table in order to obtain the forks is shown in listing 5. The scheduler does not only allow the correct execution of the philosophers but it also prevents starvation by implementing a fair scheduling strategy *e.g.* the philosopher who has asked for a fork first gets it first. In order to ensure a fair scheduling strategy, every time a philosopher tries to get two forks but either both or one of them are unavailable these forks are reserved. Reserved forks cannot be handed out to other philosophers as that would potentially lead to starvation.

The scheduler keeps track of which forks are handed out by keeping an array of `busy` forks which are initialized to be all false when the scheduler is created. Similarly, reserved forks are kept in a table called `reserved` which is local to the schedule method *e.g.* the elements in the table are initialized to false every time the schedule method is called.

```
def philosopherScheduler := scheduler: {
 def busy[5] {false};
 def schedule() {
  def reserved[5] {false};
  def letters := super.listIncomingLetters();
  foreach: { |letter|
   def philosopher := letter.receiver();
   def id1 := philosopher.id;
   def id2 := (id1%5)+1;
   if:(!busy[id1]&!busy[id2]&!reserved[id1]&!reserved[id2]) then: {
    busy[id1], busy[id2] := true;
    super.executeLetter(letter);
   } else: {
    reserved[id1],reserved[id2] := true;
   };
  } in: letters;
 };
 def leave(letter) {
  def philosopher := letter.receiver();
  def id1 := philosopher.id;
  def id2 := (id1%5)+1;
  busy[id1], busy[id2] := false;;
 };
};
```

Listing 5: Dining Philosophers Example

In the schedule method the list of pending letters is obtained with `listIncoming-Letters` and traversed in decreasing age (messages that were sent longer ago are listed first). Then the philosopher where the message is sent to (the receiver of the letter) is extracted from the letter. Each philosopher has an instance variable `id`. With this id the forks needed by this particular philosopher can be looked up. When both forks for a philosopher are free permission is granted to the philosopher to take both. The scheduler records that those forks are now taken by the philosopher by setting the corresponding booleans in the `busy` table. Otherwise, the requested forks are flagged as reserved in order to ensure a fair scheduling strategy. Because the list is ordered by decreasing age, the scheduler ensures that the philosopher which is waiting the longest for his forks will get his forks first. Finally in the leave method the forks of the philosopher which has stopped eating are set to free. This example shows that PAM's can be used for a wide range of complex coordination strategies. In the next section we show how to implement a full fledged synchronization mechanism called synchronizers.

5.4. Synchronizers

A related coordination abstraction, called synchronizers, invented by Frølund and Agha offers a declarative mechanism to deal with multi-object coordination for thread based systems [20]. As an actor can encapsulate multiple passive objects, a PAM can be used to coordinate the access to this group of objects. In this section we show that the features of synchronizers can easily be expressed in PAM to coordinate the messages sent to the objects encapsulated by an actor. The three main synchronization abstractions defined by synchronizers are: `update` to update the state of the synchronizer, `disable` to disallow the execution of certain methods (in a guard-like manner), and `atomic` to trigger several methods in parallel in an atomic manner.

To illustrate the use of these constructs consider Listing 6, which shows an example synchronizer. The aim of this synchronizer is to restrict the total amount of simultaneous requests on two objects `adm1` and `adm2`. To do so the synchronizer keeps track of the amount of simultaneous requests performed on those objects with the `prev` integer variable. Every time the method `request` is called either on `adm1` or `adm2` the `prev` value is incremented. Similarly when the method `release` is called on one of the objects the `prev` is decreased. This is implemented in the synchronizer with the `when:` *MethodPattern* `update:` *Codeblock* construct. This construct registers the *CodeBlock* to be executed whenever a method that matches the *MethodPattern* is executed. Finally, requests to both objects are

disabled when `prev` equals or exceeds `max` with the `when:` *Predicate* `disable:` *MethodPattern* construct.

```
def collectiveBound(adm1, adm2, max) { synchronizer: {
 def prev := 0;
 when: ((MethodCall: ‘request on: adm1) or (MethodCall: ‘request on: adm2)) update: {
  prev := prev + 1;
 };
 when: ((MethodCall: ‘release on: adm1) or (MethodCall: ‘release on: adm2)) update: {
  prev := prev - 1;
 };
 when: { prev >= max } disable:
 ((MethodCall: ‘request on: adm1) or (MethodCall: ‘request on: adm2));
 }; };
```

Listing 6: Synchronizer example from [20] with a PAM

The implementation of these registration constructs is shown in Listing 7. The update and disable constructs (Lines 6–7) keep track of their registrations in the `updateTable` and `disableTable` respectively (Lines 1–2). The `executeUpdates` method performs the actual invocation block given a letter that matches one of its registrations (Line 10). The `isDisabled(letter)` function returns a boolean indicating whether the given letter is currently disabled (Line 11). Similar to the `when:update:` and `when:disable:` construction the `atomic:` construction adds the method patterns to the `atomicTable` (Line 8). The `executeAtomic` method goes over the `atomicTable` and checks for each group of method patterns whether the inbox has a matching message for each of these patterns (Lines 13–18). If this is the case all these messages are executed in parallel and the executing variable is updated to keep track of the amount of messages which are being executed (Lines 15-16).

The implementation of the schedule and leave methods of the PAM is shown in Listing 8. The execution of individual messages should be prevented when the actor is executing a group of atomic messages. Similarly starting the execution of a group of atomic messages should be prevented when there are currently executing individual messages in the actor. In order to keep track of which kind of messages and how many are executing the scheduler maintains two variables, `atomic` and `executing`, which respectively indicate the execution of a group of atomic messages and the number of currently executing messages. When the schedule method is called the scheduler makes sure that it is not executing an atomic block. If this is not the case the scheduler stops in order to prevent the execution of concurrent messages during the execution of an atomic block, otherwise the scheduler proceeds. When there are no currently executing messages (indicated by the executing variable) and the scheduler can start the execution of an atomic block the

```
1  def updateTable := [];
2  def disableTable := [ default: {false} ];
3  def atomicTable := [];
4  def executing := 0;
5
6  def when: pattern update: block { updateTable := updateTable + [[pattern, block]] }
7  def when: block disable: pattern { disableTable := [[pattern, lambda]] + disableTable; }
8  def atomic: patterns { atomicTable := atomicTable + [patterns] }
9
10 def executeUpdates(letter) { dispatch: letter as: updateTable; }
11 def isDisabled(letter) { dispatch: letter as: disableTable; }
12 def executeAtomic() {
13  atomicTable.each: { |MethodPatterns|
14   if: containsAll(MethodPatterns) then: {
15    executing := executeAll(MethodPatterns);
16    return true;
17   }
18  }
19  return false;
20 }
```

Listing 7: Synchronizers registration constructs implemented in AmbientTalk

variable `atomic` is changed accordingly. Otherwise the scheduler walks over the mailbox of incoming letters and executes all the letters which are not disabled. For each letter which is not disabled the scheduler starts it's execution, updates the executing variable by incrementing it by one and executes all the updates. The leave method of the scheduler decrements the executing variable by one for each leaving letter and sets the atomic variable to false when the executing counter equals 0. Synchronizers offer the programmer a declarative mechanism for coordinating multiple objects, however they are more limited than our PAM abstraction: fairness cannot be specified at the application level; history-based strategies must be manually constructed; and finally, although synchronizers encapsulate coordination, their usage has to be explicit in the application, requiring intrusive changes to the existing code. Also, synchronizers explicitly reference method names: their reuse potential is therefore more limited than PAM, where the method categories allow for more abstraction.

6. AmbientTalk Implementation

Parallel Actor Monitors are implemented in AmbientTalk using reflection. AmbientTalk is a prototype-based actor language with a strong reflective layer. This reflective layer has been implemented by the design principles of so called *mirrors*. A mirror is a special kind of object that encapsulates a number of op-

```
1  def schedule() {
2   if: !atomic then: {
3    if: (executing == 0 & executeAtomic()) then: {
4     atomic := true;
5    } else: {
6     listIncomingLetters().each: { |letter|
7      if: ( (!isDisabled(letter)).and: { !inAtomic(letter) }) then: {
8       executeUpdates(letter);
9       executing := executing + 1;
10      super.executeLetter(letter);};};};};};
11
12 def leave() {
13  executing := executing - 1;
14  if: (executing == 0) then: { atomic := false}
15 };
```

Listing 8: Schedule method for implementing Synchronizers with PAM

erations which are called by the underlying implementation in order to allow a program to alter the default semantics of a particular feature of the system. The specific order and time when these meta operations are invoked are described in the meta object protocol. For example, in AmbientTalk the message meta-object protocol describes that the `receive` method of a mirror is called every time an asynchronous message is sent to the base level object of that mirror. Mirrors where proposed by Bracha and Ungar [23] in order to provide structural introspection and intercession. The AmbientTalk model has extended the concept of mirrors in order to also perform behavioral intercession [24]. Mirrors in the AmbientTalk model can be created in the base level and attached to an actor or an object upon creation time. From there on the meta-object protocol will be applied to the newly attached mirror. PAM has been implemented as an extension of this existing mirror based reflective layer.

The second part of the implementation consists of a global thread-pool which is used for the execution of the letters. This has the advantage that we can avoid the startup times for each thread after the initialization of the thread-pool. One of the key points of our implementation is that a PAM is a *thread-less* scheduler. By re-using the threads in the thread-pool for scheduling we can avoid unnecessary thread-switches opposed to a strategy where the scheduler is a thread on its own.

7. UrbiFlock a Use Case for PAM

The AmbientTalk language has been primarily designed to ease the development of so called Ambient Applications running on mobile phones. The communication in these applications is conducted in a peer to peer fashion without

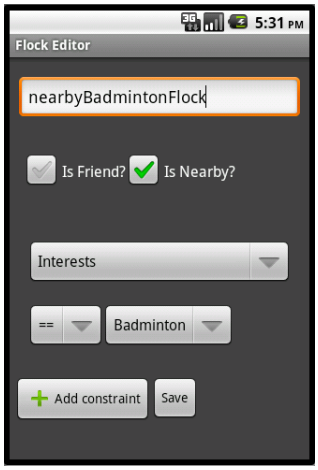

Figure 3: Screenshot of the UrbiFlock UI to add a new nearby badminton flock.

any infrastructure such as a centralized server. UrbiFlock [25, 26] is a framework implemented on top of AmbientTalk for the development of a brand new kind of application which enables social networking on the move. Central in the UrbiFlock framework is the concept of a Flock, users can group friends in different categories (Flocks). Flocks differentiate themselves from normal user groups as the content of a Flock is automatically updated in order to reflect contextual changes in the environment. For example a user can define a Flock of all nearby users who like badminton. A screenshot of how an end user can define this flock is shown in figure 3. From then on the badminton flock will be periodically updated in order to reflect the changes in context. For example when users move in and out of communication range.

Because in Ambient Applications the computation of such a Flock is necessarily conducted on the mobile phone itself (we cannot rely on a centralized server) the derivation of which user belongs to which flock has to be computed efficiently. However, as seen in the introduction using the actor model adopted in AmbientTalk forces the programmer to adopt a sequential execution. Recent advances in mobile computing have lead to the development of mobile phones which have multicore processors[4]. While actor applications such as UrbiFlock will certainly benefit from these new architectures, the use of the actor model itself will prevent the parallel execution of matching multiple users at the same time.

[4]http://tinyurl.com/38pkzd4

Therefore, we have applied the techniques developed in PAM in order to speed up the matching phase. As we do not have access to these new multicore mobile phones we have extracted the matching phase of a Flock and conducted our experiments on a dual core laptop. The derivation of the Flock that we have benchmarked computes all the persons who's partner is also in the neighborhood. This means that whenever a new person appears and exchanges his profile the content of the Flock has to be recomputed. In our test we have benchmarked the

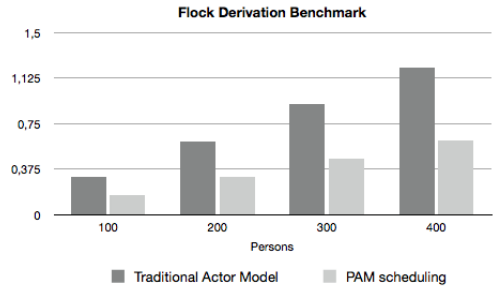

Figure 4: Speedup for the flock derivation by using a PAM.

behavior of a person with three hundred friends in his friend list entering a large hall. The benchmark starts when the new content of the Flock is computed and ends when all the couples have been found. We have benchmarked the derivation of this Flock both for a serial execution and parallel execution (enabled by the reader-writer pam shown in listing 1). The x-axis consists of the number of people joining the flock. For a serial execution we already notice that the derivation time of the couples flock will take over a second. Note that these computations are conducted on an Intel Core 2 Duo with a processor speed of 1.8 GHz running Mac OS X (10.5.8) and thus that the expected derivation times on a mobile phone will take more time. By plugging in the reader–writer PAM we see that we can obtain a two times speedup in almost all the cases. With hundred people the speedup is already 1.88 and at four hundred this speedup increases to 1.96. In the next section we give more details about the conducted micro benchmarks.

8. Evaluation

In this section an evaluation of the contributions of using PAM over current practices is shown, for each contribution a discussion of the advantages and disadvantages of current practices is given. An overview of our evaluation is shown in figure 2.

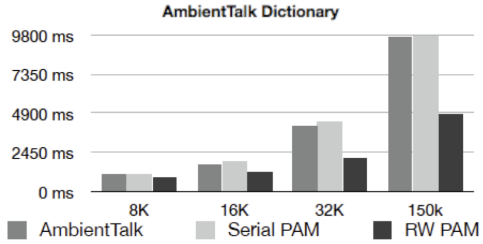

Figure 5: Processing time for the dictionary in the AmbientTalk implementation.

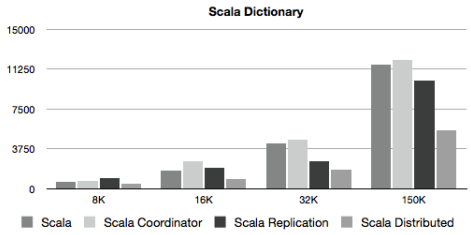

Figure 6: Processing time for the dictionary in the Scala implementation.

8.1. Efficiency

A PAM makes it possible to take advantage of parallel computation for improved scalability. Benchmarks of our prototype implementations suggest speedups that are almost linear to the number of processors available. In Figure 5 micro-benchmarks show both the overhead of using a mutual exclusion scheduler PAM (Listing 1) compared to plain AmbientTalk and the possible speedup when using the reader-writer PAM scheduler (Listing 3).

The results depicted were obtained on an Intel Core 2 Duo with a processor speed of 1.8 GHz running Mac OS X (10.5.8). We measured the processing time of reading one hundred items from a dictionary actor[5] of varying size. These results show the low overhead of PAM (< 6%) and the expected speedup (1.9x for the 32K and 150K dictionary), taking almost full advantage of the 2 cores.

In figure 6, the same dictionary example implemented in Scala [7] is shown. Four different situations are benchmarked in this test. The first benchmark `Scala`, measures the situation where one client sends hundred read messages to a dictionary actor. In the second benchmark `Scala Coordinator`, a coordinator actor is placed between the client and the dictionary actor. This coordinator simply forwards the messages to a single dictionary actor. This is similar to the situation of a serial PAM implementation. As expected there is only a small overhead. The `Scala Replication` benchmark measures the performance where the coordinator actor forwards the read messages to a group of worker actors which all receive a *replica* of the dictionary. This simulates the situation where a complex data-structure can not be partitioned over a number of workers and thus needs to be replicated in order to speed up reads. As can be seen, only in the situation of

[5]The dictionary is implemented as an association list of key value pairs.

a dictionary of size 32K a reasonable speedup is achieved (1.6). When the dictionary becomes bigger (150K) the performance degrades to a 1.14 speedup. In the final benchmark the workers all receive *an equal part of the dictionary* and the messages are *perfectly balanced* over the dictionaries. Only in this case, similar speedups as with the PAM reader-writer PAM scheduler could be achieved. Because these speedups could only be achieved under perfectly partitioned and balanced dictionaries Table 2 indicates a +/- sign for the traditional actor model.

When using a shared data structure, transactors or when making use of a thread library within the actor, similar speedups can be achieved. However, all these approaches fail to do so in a structured way as shown in the next sections.

8.2. Modularity

A PAM is a modular, reusable scheduler that can be parameterized and plugged into an actor to introduce intra-actor parallelism *without modification* of the original code. With approaches that allow shared data structures to be introduced, programmers need to adjust the code with a locking strategy or introduce a special coordinator actor which governs the access to the shared data structure. Clearly the use of a thread library through symbioses requires intrusive changes to the code. In the Scala example, modularity is not preserved as the worker actors can not be encapsulated in the coordinator actor and possibly leak to other parts of the program. Further, the code to create the clients needs to be adjusted in order to pass the coordinator actor instead of a concrete dictionary actor. With transactors, the body of the actor has to be adjusted in order to spawn new transactions and only transactional references are capture by the transaction system.

8.3. Locality

Binding a PAM to an actor *only* affects the parallel execution of messages inside that single actor. Introducing shared data structures has an impact on the whole actor program, failing to request access to the coordinator actor will lead to errors. In the implementation of the dictionary example in Scala locality is an issue because direct access to one of the worker actors can potentially affect the whole program. Transactors are not limited to a single actor and thus can potentially affect the behavior of other actors, as this is very unlikely to happen accidentally the table shows a (-/+).

8.4. Abstraction

A PAM is expressed at the same level of abstraction as actors: the scheduling strategy realized by a PAM is defined in terms of a message queue, messages, and

System	Efficiency	Modularity	Locality	Abstraction
Trad. Actors	-/+	-	-	+
Transactors	+	-	-/+	-
Symbiosis	+	-	+	-
Shared Data	+	-	-	-
PAM	+	+	+	+

Table 2: Comparison of the core contributions of PAM against current state of the art.

granting permissions to execute. With the exception of the traditional actor model all other solutions require the programmer to think in a different paradigm.

9. Conclusion

In order to address data-task entanglement resulting from the strict restrictions of the actor model, we have proposed the model of Parallel Actor Monitors. Using PAM, there can be intra-actor parallelism, thereby leading to better scalability by making it possible for a single actor to take advantage of the parallelism offered by the underlying hardware. PAM offers a particularly attractive alternative to introduce parallelism inside actors, because it does so in a modular, local, and abstract manner: modular, because a PAM is a reusable scheduler, specified separately; local, because only the internal activity of an actor is affected by using a PAM; abstract, because the scheduler is expressed in terms of (categories of) messages, queues and granting permission to execute. Benchmarks on dual and oct core machines confirm the expected speedups.

References

[1] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[2] O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare (Eds.), Structured programming, Academic Press Ltd., London, UK, UK, 1972.

[3] R. Diaconescu, Object based concurrency for data parallel applications: Programmability and effectiveness, Ph.D. thesis, Norwegian University of Science and Technology, Department of Computer and Information Science (2002).

[4] R. K. Karmani, A. Shali, G. Agha, Actor frameworks for the jvm platform: a comparative analysis, in: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, ACM, 2009, pp. 11–20.

[5] A. Yonezawa (Ed.), ABCL: An Object-Oriented Concurrent System, Computer Systems Series. MIT Press, 1990.

[6] J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent Programming in ERLANG, Prentice Hall, 1996.

[7] P. Haller, M. Odersky, Actors that unify threads and events, in: Proc. of the 9th inter. conf. on Coordination models and languages, COORDINATION'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 171–190.

[8] S. Srinivasan, A. Mycroft, Kilim: Isolation-typed actors for java, in: Proceedings of the 22nd European conference on Object-Oriented Programming, ECOOP '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 104–128.

[9] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Grid Computing: Software Environments and Tools, Springer-Verlag, 2006, Ch. Programming, Deploying, Composing, for the Grid.

[10] M. Stiegler., The E language in a walnut, www.skyhunter.com/marcs/ewalnut.html.

[11] C. Varela, G. Agha, Programming dynamically reconfigurable open systems with SALSA, ACM SIGPLAN Notices. OOPSLA'2001 36 (12) (2001) 20–34.

[12] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter, Ambient-oriented Programming in Ambienttalk, in: D. Thomas (Ed.), ECOOP, Vol. 4067, Springer, 2006, pp. 230–254.

[13] R. Hickey, Message passing and actors, Clojure online documentation, http://clojure.org/state.
URL `http://clojure.org/state`

[14] J. Gosling, B. Joy, G. Steele, The Java Language Specification, GOTOP Information Inc., 1996.
URL `citeseer.ist.psu.edu/gosling96java.html`

[15] J. Schäfer, A. Poetzsch-Heffter, Jcobox: generalizing active objects to concurrent components, in: Proc. of the 24th European conf. on Object-oriented

programming, ECOOP'10, 2010, pp. 275–299.

[16] E. B. Johnsen, J. C. Blanchette, M. Kyas, O. Owe, Intra-object versus inter-object: Concurrency and reasoning in creol, Electron. Notes Theor. Comput. Sci. 243 (2009) 89–103.

[17] C. Fournet, G. Gonthier, The reflexive cham and the join-calculus, in: POPL '96, ACM, New York, NY, USA, 1996, pp. 372–385.

[18] N. Benton, L. Cardelli, C. Fournet, Modern concurrency abstractions for c#, ACM Trans. Program. Lang. Syst. 26 (5) (2004) 769–804.

[19] G. S. Itzstein, M. Jasiunas, On implementing high level concurrency in java, in: In Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference, Springer, 2003, pp. 151–165.

[20] S. Frølund, G. Agha, A language framework for multi-object coordination, in: ECOOP '93, Springer-Verlag, London, UK, 1993, pp. 346–360.

[21] D. Caromel, L. Mateu, G. Pothier, É. Tanter, Parallel object monitors, Concurrency and Computation—Prac. and Exp. 20 (12) (2008) 1387–1417.

[22] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (8) (1975) 453–457.

[23] G. Bracha, D. Ungar, Mirrors: Design principles for meta-level facilities of object-oriented programming languages, in: Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications, 2004, pp. 331–343.

[24] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, W. De Meuter, Mirror-based reflection in ambienttalk, Softw. Pract. Exper. 39 (7) (2009) 661–699.

[25] A. Carreton, D. Harnie, E. Boix, C. Scholliers, T. Van Cutsem, W. De Meuter, Urbiflock: An experiment in dynamic group management in Pervasive social applications, in: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, IEEE, 2010, pp. 250–255.

[26] E. Gonzalez Boix, A. Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, T. D'Hondt, Flocks: Enabling dynamic group interactions in mobile social networking applications, in: 26th Symposium On Applied Computing (SAC) Track on Mobile Computing and Applications (MCA), 2011.